# C#: Future Directions in Language Innovation

PDC05 DEVELOPER POWERED

Anders Hejlsberg
TLN307
Technical Fellow
Microsoft Corporation

Microsoft

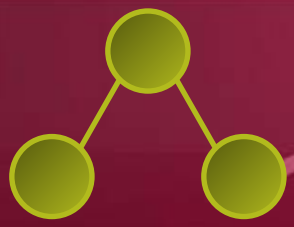
# The LINQ Project

FUTURE TECHNOLOGIES

C# | VB | Others...

## .NET Language Integrated Query

Standard Query Operators | DLinq (ADO.NET) | XLinq (System.Xml)



Objects

SQL

WinFS


XML

# C# 3.0 Design Goals

FUTURE TECHNOLOGIES

- Integrate objects, relational, and XML
- Build on foundation laid in C# 1.0 and 2.0
- Run on the .NET 2.0 (“Whidbey”) CLR
- Don’t tie language to specific APIs
- Remain 100% backwards compatible

DEMO

# Language Integrated Query

PDC05
DEVELOPER POWERED

# C# 3.0 Language Innovations

```
var contacts =
    from c in customers
    where c.State == "WA"
    select new { c.Name, c.Phone };
```

Query expressions

Local variable type inference

```
var contacts =
    customers
    .Where(c => c.State == "WA")
    .Select(c => new { c.Name, c.Phone });
```

Lambda expressions

Extension methods

Anonymous types

Object initializers

# Lambda Expressions

```
public delegate bool Predicate<T>(T obj);

public class List<T>
{
    public List<T> FindAll(Predicate<T>
}

List<Customer> customers =

List<Customer> x = customers.FindAll(
    delegate(Customer c) { return c.State ==
"WA"; }
);

List<Customer> x = customers.FindAll(c => c.State
== "WA");
```

Explicitly typed

Statement context

Implicitly typed

Expression context

# Lambda Expressions

```
public delegate T Func<T>();
public delegate T Func<A0, T>(A0 arg0);
public delegate T Func<A0, A1, T>(A0 arg0,
A1 arg1);
...
```

```
Func<Customer, bool> test = c => c.State
== "WA";
```

```
double factor = 2.0;
Func<double, double> f = x => x * factor;
```

```
Func<int, int, int> f = (x, y) => x * y;
```

```
Func<int, int, int> comparer =
    (int x, int y) => {
        if (x > y) return 1;
        if (x < y) return -1;
        return 0;
    };
```

# Queries Through APIs

```
public class List<T>
{
   public List<T> Where(Func<T, bool> predicate) { ... }
   public List<S> Select<S>(Func<T, S> selector) { ... }
   ...
}
```

Query operators are just methods

```
List<Customer> customers =
GetCustomerList();
```

```
List<string> contacts =
   customers.Where(c => c.State == "WA").Select(c
```

Methods compose to form queries

But what about other types?

Declare operators in all collections?

What about arrays?

Type inference figures out <S>

# Queries Through APIs

```
public static class Sequence
{
    public static IEnumerable<T> Where<T>(IEnumerable<T>
source,
        Func<T, bool> predicate) { ... }

    public static IEnumerable<S> Select<T,
S>(IEnumerable<T> source,
        F
    ...
}
```

Query operators are static methods

```
Customer[] customers = GetCustomerArray(
```

Huh?

```
IEnumerable<string> contacts = Sequence.Select(
        Sequence.Where(customers, c => c.State ==
"WA"),
        c => c.Name);
```

Want methods on IEnumerable<T>

# Extension Methods

```
namespace System.Query
{
    public static class Sequence
    {
        public static IEnumerable<T> Where<T>(this
IEnumerable<T> source,
            Func<T, bool> predicate) { ... }

        public static IEnumerable<S> Select<T, S>
IEnumerable<T> source,
            Func<T, S> selector
    }
}
```

Extension methods

```
using System.Query;
```

Brings extensions into scope

obj.Foo(x, y)
↓
XXX.Foo(obj, x, y)

```
IEnumerable<string> contacts =
    customers.Where(c => c.State == "WA").Select(c
=> c.Name);
```

IntelliSense!

DEMO

# Lambda Expressions and Extension Methods

PDC05
DEVELOPER POWERED

# Object Initializers

```
public class Point
{
    private int x, y;

    public int X { get { return x; } set { x =
value; } }
    public int Y { get { return y; } set { y =
value; } }
}
```

```
Point a = new Point { X = 0, Y
= 1 };
```

Field or property assignments

```
Point a = new Point();
a.X = 0;
a.Y = 1;
```

# Object Initializers

```
public class Rectangle
{
   private Point p1 = new Point();
   private Point p2 = new Point();

   public Point P1 { get { return p1; } }
   public Point P2 { get { return p2; } }
}
```

Embedded objects

Read-only properties

```
Rectangle r = new Rectangle {
   P1 = { X = 0, Y = 1 },
   P2 = { X = 2, Y = 3 }
};
```

No “new Point”

```
Rectangle r = new
Rectangle();
r.P1.X = 0;
r.P1.Y = 1;
r.P2.X = 2;
r.P2.Y = 3;
```

# Collection Initializers

Must implement ICollection<T>

```
List<int> powers = new List<int> { 1, 10, 100,
1000, 10000 };
```

```
List<int> powers = new
List<int>();
powers.Add(1);
powers.Add(10);
powers.Add(100);
powers.Add(1000);
powers.Add(10000);
```

# Collection Initializers

```
public class Contact
{
    private string name;
    private List<string> phoneNumbers = new List<string>();

    public string Name { get { return name; } set { name =
value; } }
    public List<string> PhoneNumbers { get { return
phon
}
```

```
List<Contact> contacts = new List<Contact> {
    new Contact {
        Name = "Chris Smith",
        PhoneNumbers = { "206-555-0101", "425-882-
8080" }
    },
    new Contact {
        Name = "Bob Harris",
        PhoneNumbers = { "650-555-0199" }
    }
};
```

# Local Variable Type Inference

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new
Dictionary<int,Order>();
```

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

"var" means same type as initializer

# Anonymous Types

```
public class Customer
{
    public string Name;
    public Address Address;
    public string Phone;
    public List<Order> Orders;
    ...
}
```

```
public class Contact
{
    pub
    pub
}
```

```
class ???
{
    public string
Name;
    public string
Phone;
}
```

```
Customer c = GetCustomer(...);
Contact x = new Contact { Name = c.Nam      hone =
c.Phone };
```

```
Customer c = GetCustomer(...);
var x = new { Name = c.Name, Phone = c.Phone };
```

```
Customer c = GetCustomer(...);
var x = new { c.Name, c.Phone };
```

Projection style initializer

# Anonymous Types

```
var contacts =
    from c in customers
    where c.State == "WA"
    select new { c.Name, c.Phone
```

IEnumerable<???>

```
var contacts =
    customers.
    .Where(c => c.State == "WA")
    .Select(c => new { c.Name,
Phone });
```

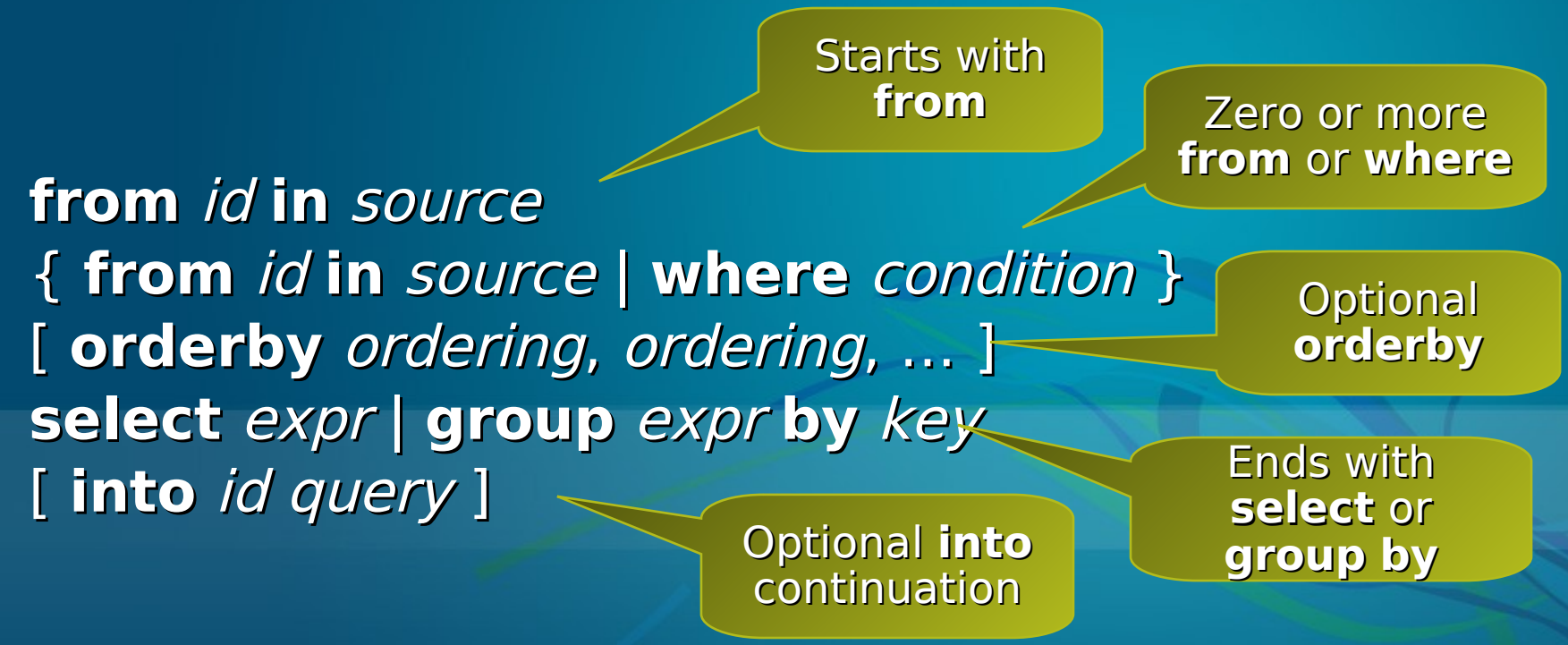
???

```
foreach (var c in contacts) {
    Console.WriteLine(c.Name);
    Console.WriteLine(c.Phone);
}
```

```
class ???
{
    public string
Name;
    public string
Phone;
}
```

# Query Expressions

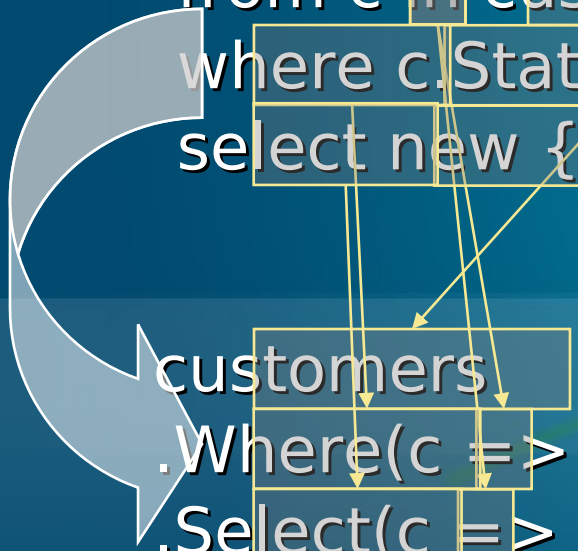
- Language integrated query syntax



# Query Expressions

- Queries translate to method invocations
  - Where, Select, SelectMany, OrderBy, GroupBy

```
from c in customers
where c.State == "WA"
select new { c.Name, c.Phone };
```



```
customers
.Where(c => c.State == "WA")
.Select(c => new { c.Name, c.Phone });
```

# Expression Trees

```
public class Northwind: DataContext
{
    public Table<Customer>
Customers;
    public Table<Order> Orders;

}
```

```
Northwind db = new Northwind(...);
var query = from c in db.Customers where c.State ==
"WA" select c;
```

```
Northwind db = new Northwind(...);
var query = db.Customers.Where(c => c.State
== "WA");
```

```
public class Table<T>: IEnumerable<T>
{
    public Table<T> Where(Expression<Func<T, bool>>
predicate);
    ...
}
```

How does this get remoted?

Method asks for expression tree

System.Expressions.
Expression<T>

# Expression Trees

- Code as Data

```
Func<Customer, bool> test = c => c.State
== "WA";
```

```
Expression<Func<Customer, bool>> test = c => c.State
== "WA";
```

```
ParameterExpression c =
    Expression.Parameter(typeof(Customer), "c");
Expression expr =
    Expression.EQ(
        Expression.Property(c,
typeof(Customer).GetProperty("State")),
        Expression.Constant("WA")
    );
Expression<Func<Customer, bool>> test =
    Expression.Lambda<Func<Customer, bool>>(expr, c);
```

DEMO

# LINQ Tech Preview

PDC05
DEVELOPER POWERED

# C# 3.0 Language Innovations

- Lambda expressions
- Extension methods
- Local variable type inference
- Object initializers
- Anonymous types
- Query expressions
- Expression trees

```
c =>
c.Name
```

```
static void Dump(this
object o);
```

```
var x = 5;
```

```
new Point { x = 1, y
= 2 }
```

```
new { c.Name,
c.Phone }
```

```
from ... where ...
select
```

```
Expression<T
>
```

# More Information

| Wednesday | Thursday | Friday |
|---|---|---|
| **TLN306** - LINQ Overview<br>1:45 PM - 3:00 PM<br>Halls C & D | **TLN308** - VB 9.0<br>10:00 AM - 11:15 AM<br>Room 411 | **DAT323 (R)** - DLinq for SQL<br>8:30 AM - 9:45 AM<br>Room 408 AB |
| **TLN307** - C# 3.0<br>3:15 PM - 4:30 PM<br>Halls C & D | **TLN307 (R)** - C# 3.0<br>2:15 PM - 3:30 PM<br>Room 402 AB | **DAT324** - XLinq<br>10:30 AM -11:45 AM<br>Room 408 AB |
| | **DAT323** - DLinq for SQL<br>2:15 PM - 3:30 PM<br>Room 152/153 (Hall F) | **PNL11** - LINQ Panel<br>1:00 PM - 2:30 PM<br>Room 152/153 (Hall F) |
| | **DAT312** - DLinq for WinFS<br>5:00 PM – 6:15 PM<br>Room 515 AB | |
| | **Ask the Experts**<br>6:30 PM - 9:00 PM<br>Main Hall | |

http://msdn.microsoft.com/netframework/future/linq/

Microsoft®

Your potential. Our passion.™